ray tracing

HoloPov: A previsualization program for holographers

Kaveh Bazargan
kaveh@focalimage.com

About the author

Kaveh's interest began with seeing the Royal Academy of Arts exhibition of holograms in London in 1976. He was studying physics at Imperial College London, and continued to complete a PhD in 'display holography'. His passion remains the quest to achieve ultimate realism with holography.

Introduction

Over the years I have often wanted to be able to simulate the recording and reconstruction process in holograms. I am particularly interested in display holograms, as opposed to HOEs. During my studies I did write little programs for specific cases, but not a general purpose program. There are two parts to such a program: first to calculate the reconstructed image, having fed in the the recording data, and second, to present this in a nice graphical form, preferably three-dimensional.

Recently I discovered *POV-ray* (http://www.povray.org), a 3D image rendering engine, and realised it was the ideal tool for the job. Please see the the amazing images people have created with it. *POV-ray* is free, platform-independent (Linux, Unix, Macintosh, Windows), and produces images equal to renderers at *any* price. The beauty of using *POV-ray* is that not only is it a superb 3D rendering engine, but it is also an excellent programming language. Never calling myself a *real* programmer, I slowly put some equations in, got results, and soon got carried away! Slowly I built in more capability. The more I work on it the more possibilities I see to simulate the holographic process. I can now see that we can simulate full color holograms, 'pseudocolour' holograms, and rainbow holograms. At the moment it is limited to transmission holograms, but I hope to extend it to reflection holograms too.

The basic equations used are those of Champagne (*J. Opt. Soc. Amer.* **57** (1967) p. 51). In this document I'll not go into the equations, but just the functionality of the program. This version is really still being worked on, and there will be bugs, so I do appreciate feedback.

License

HoloPov is released under the GNU LGPL license (see http://www.fsf.org/licenses/licenses.html#LGPL). The code is therefore 'open' and you

can modify it and use it free of charge, and to redistribute it, within the restrictions of the license. In the spirit of Free Software, I encourage people to test and modify the code to improve and debug it, and help serve the holographic community.

Requirements

You should be running *POV-ray* on your computer. I have used version 3.5. The files should run with any installation of *POV-ray* above 3.5. *POV-ray* runs on most platforms and is device independent.

Examples

To see some examples of the kind of output produced, please see my other article, "White light transmission holograms"

Availability

HoloPov is available for download here.

File structure

There are three files in the package, `holopov.pov` and `holopov.inc`, and `optics.inc`. You will also need to get `CIE.inc` from `http://www.ignorancia.org/zips/lightsys4.zip`. `holopov.pov` is the file that is run through *POV-ray*. It will `#include holopov.inc` which will in turn `#include` several other files. Most of these are standard files distributed with *POV-ray*, but `optics.inc` is a library file of mine, and `CIE.inc` is a third party file which, amongst other things, works out the approximate color for a given wavelength.

The coordinate system

We use the coordinate system of *POV-ray*, i.e. a *left-handed* cartesian system. The hologram is always in the x–y plane, centered at the origin. If you imagine your computer screen as being the hologram, with the x-axis pointing to the right, and y pointing up, then the z-axis will be pointing into the screen. The center of the screen is `<0,0,0>`

I have chosen to have the observer in the $z < 0$ region, and other points in the $z > 0$. But I think that any point can have any coordinates. If the formulae have been coded properly, then this should be the case *I think*. I have not dared test this yet!

Overall concept

The basic idea is that you dial in the recording and the reconstruction parameters, and the program will work out the position of the image. By 'parameters', I mean the following:

- position of the object
- position of the recording source
- position of the reconstruction source
- position of the observer
- recording wavelength
- reconstruction wavelength

All these values are entered into the main file, i.e. `holopov.pov`.

The object

These are the parameters that define the object:

- `obj_dist`
- `obj_angle`
- `obj_side_angle`
- `obj_theta`
- `object_size`
- `grid_sep`

After a lot of thought, I decided that spherical polar coordinates were the best way to enter the data for the object, at least for me.

`obj_dist` is the distance of the center of the object from the origin, i.e. from the center of the plate. This value is always positive.

`obj_angle` is the angle of the object in the y–z plane, i.e. above or below the horizon.

`obj_side_angle` is the angle of the object in the x–z plane, i.e. the angle if we were looking directly from above.

`obj_theta` is the angle around the y-axis. This the angle one would see if one were looking down the y-axis from above.

To have an object in the observer space, i.e. $z < 0$, we can use values of `obj_theta, obj_angle` $> 90°$ or $< -90°$.

`object_size` needs some explanation. I have defined an object to be not just a point, but a 3-dimensional grid of points. The three coordinates in `object_size` define the extent of the image in each axis. For example `<2,2,2>` means a cube, with dimension `2` on each side. `grid_sep` defines the distance between successive grid points. The smaller the value of `grid_sep`, the more grid points will be present. If `object_size` is set to `<2,0,2>`, the grid will effectively be a sheet, with zero height.

Please note that values corresponding to object distance and angle refer to the average object position, i.e. the center of the object.

Recording geometry

Here are the values to set:

- `ref_dist`
- `ref_angle`
- `ref_side_angle`
- `ref_theta`
- `lambda_r`

Again we use spherical polar coordinates. `lambda_r` is the wavelength used for recording the hologram. Again we should be able to use angles $> 90°$ and $< -90°$ to have, for example, a converging beam. I need to do a bit more thinking on this. For example, how to distinguish between a converging beam and a diverging beam, both referring to the same point in space. I hope that we can have a sign system that caters for all combinations.

Reconstruction geometry

Here are the corresponding values in reconstruction:

- `rec_dist`
- `rec_angle`
- `rec_side_angle`
- `rec_theta`
- `lambda_c`

Observer

These values define the observer:

- `obs_dist`
- `obs_angle`
- `obs_side_angle`

I have been using angles of zero, and a negative observer distance to signify an observer on the z-axis, but to be consistent with above.

Camera settings

These are the more or less standard settings used in *POV-ray* for the camera:

- `camera_loc` – The camera position. Presently in cartesian coordinated, but might be better to change it to polar in future releases.
- `camera_look` – Center of the rendered image.
- `camera_angle` – As in the normal *POV-ray* definition. To zoom in, we decrease this value.

H1 and H2

I have managed to simulate the effect of having a master hologram, or as it is normally referred to, the H1. The hologram doing the final imaging, I will always refer to as the H2. The dimensions of the two plates are set by:

- `h1_width`
- `h1_height`
- `h2_width`
- `h2_height`

The main reason for setting the dimensions of the two plates is to take into account the image cut-off, or vignetting, that occurs as the viewer moves around. We also need a value to define the position of the H1 relative to H2. We could of course give the distance between the two, but I have opted to define `h1_sep`, which is the distance from the center of the object to that of the H1. So as we change the position of the object, all other values being equal, we are also implicitly changing that of the H1.

Options

I have put in a lot of options which have a bearing on how the final image is drawn. These are set as parameters that are tested later in `holopov.inc`. A value of 0 implies the value is false, any other value sets the value to true. Here are the list of options now available:

- `object_points_true` – Draws a sphere at each object grid point. The color is a color approximating to the recording wavelength. This is achieved by the `Wavelength()` macro in `CIE.inc` (see the file for details). The radius of the sphere is determined by `object_sphere_rad` in `holopov.inc`.
- `object_grid_true` – Joins up the object grid points with cylinders, color as above. The radius of the cylinder is set by `grid_cylinder_rad` in `holopov.inc`.
- `image_points_true` – As with `object_points_true`, but for image points. Color corresponds to the reconstruction wavelength. Radius is set by `image_sphere_rad`. I have set the object and image radii separately.
- `image_grid_true` – As with `object_grid_true`, but for the image. The radius of the cylinder is set by `grid_cylinder_rad`.
- `ref_beam_true` – Draws four lines from the reference point to the four corners of the recording plate. The 'lines' are actually cones which taper to zero at the reference point. The maximum radius of the cone is equal to `ray_radius`. Color approximates to the reference wavelength.
- `rec_beam_true` – As with `ref_beam_true`, but for the reconstruction beam. Color as expected.
- `obj_beam_true` – Draws a cylinder from the center of the plate to the position of the observer. Color that of the recording wavelength, radius set by `ray_radius`.
- `image_beam_true` – Joins the center of the image to the observer position. The section from the observer to the plate is a solid cylinder, radius determined by `ray_radius`, and the section from the plate to the image is drawn as a dotted line. Paramters for this are `dotted_space` and `dotted_ray_rad`.
- `eye_true` – Draws a pair of eyes, a bit evil looking at present, at the observer position, and looking at the center of the image.
- `plate_true` – Draws the recording plate, the dimensions of which are determined by `Plate`, in `holopov.inc`. I have left this in the `.inc` file,

as I have not had to change it much, but it can be moved to the main `.pov` file.

- `vignette_true` – This option will draw only those parts of the image which would be visible with the geometry described, i.e. the image will be cut off at the edges of the plate.

- `disp_comp_true` – Uses 'dipersion compensation'. This means it disregards the value set for `rec_angle`, and adjusts it so that the image is reconstructed as close to the object position as possible. In other words, it compensates for lateral chromatic dispersion.

H1 and H2 options

We'll list the H2 options first, as the the H2 is always present, but H1 is optional:

- `h2_visible` – This just means that the H2 plate is drawn in the final rendering. Sometimes it might be best not to draw it.

- `h2_vignette_true` – This will simulate the vignetting effect, or the cut-off, of the h2. In other words, an image point is drawn only when the straight line from that point to the eye intersects with the H2. This works for real and virtual images.

- `h2_vignette_visible` – This draws a faint volume in space, rather like a distorted pyramid, showing how the vignetting works. It is included for educational purposes.

- `h1_visible` – As above, but shows the H1. I have only tried this in viewer space, i.e. a real image of the H1 projected, but it should work in the case of a virtual image too.

- `h1_vignette_true` – As with H2, but this is far more important for holgraphers, as it determines the viewing angle for the image.

- `h1_vignette_visible` – As with H2.

Camera options

- `camera_observer_true` – The camera is at the observer position (between the two eyes!). I have added `0.1*y` to its value later, so that the camera is not blocked by the image or object beams cylinders, if these are present.

- `look_at_image_true` – Always keeps the center of the image in the center of the picture.

- `orthographic_true` – Uses orthographic viewing. Useful if looking down at one of the axes. Actually, we are using a 'cheat' orthographic view, because true orthographic is not compatible with `screen.inc` which we need in order to put the data in the corner of the image. In our case we set a high value for the distance of the camera, and adjust the camera angle to a very small one. If you are interested, see `#if (orthographic_true)...` in `holpov.inc`.

Multiple images

Quite often we want to superimpose in the same picture, the result of varying one of the values, for example, observer position, reconstruction wavelength, etc. I have defined a variable called `repeat_image`. This can be set to a value which determines which parameter is to be varied. For example, we can say `#declare repeat_image = Lambda_c` to vary the reconstruction wavelength. Note the upper case first letter to distinguish this 'choice' from the actual value, which is `lambda_c`. (I have previously set `Lambda_c` to a number, so this is just a trick to make it easier to remember what is to be varied.)

In `holopov.pov` I have inserted and commented out all the possible variables, so these just need to be uncommented. If nothing is to be varied, then we choose `None`.

The value `repeat_amplitude` determines the 'amplitude' of the variation around the central value set previously. So, for example, if `lambda_c` has been set to `550`, and `repeat_amplitude` is set to `150`, then the range for repetition will be from 400 to 700nm. The number of steps is determined by `number_of_steps`. A value of `2` will result in two variations, one for 400 and one for 700nm.

Animation

In addition to multiple images in the same output, we can animate the scene by varying any of these parameters. Using multiple images and animations are independent, so it is possible to have a multiple image that is animated too. For instance, we can show the effect of using different wavelengths to reconstruct in the same scene, using multiple images, then animate that whole scene by moving the observer in an arc, by varying `obs_theta`.

The equivalent of `repeat_amplitude` in animation is the `animate` variable, and it is set in the same way as in multiple images.

There is one more parameter that determines how the frames are distributed through time: if `sinusoidal_true` is set to zero, then the frames are evenly distributed through time. Otherwise, the distribution is sinusoidal. In other words, the variation is fast near the central value, and slows down

towards the extreme values. When the final animation is 'looped back and forth', this gives a more natural, smooth motion.

Running the animation

Please note that a command for running the animation must be given, otherwise the animation settings will be ignored. The precise method for running the animation depends on your installation. You should either choose animation through the user interface, or use a command line if you are using Linux or the command line version on Mac OS X (which is what I use). In any case, the most important point is that the `clock` variable must go from `-1` to `1`. Other options, such as total number of frames, resolution, etc, are set according to your preference.

Of course *POV-ray* can only produce the individual frames for an animation. You will need to use a utility to convert these to a true movie file. I use QuickTime Pro, and it works pretty well. The best way to look at the animation is to loop the animation back and forth continuously. In QT Pro, choose 'loop back and forth' from the Movie menu.

Other comments

Data output

The reason for including `screen.inc` is to allow data to be output to the file. This works very nicely. Presently only a few values are output, and there is no choice as to what to print. I need to spruce this up a bit.

The maths

This was a bit more complex than I thought. The basic equations are those of Champagne of course, but there is a basic complication in display holography which is not normally mentioned. In a conventional optical system, the 'pupil' is normally given, and the principal ray is that which goes through the center of the pupil. So we can apply the equations and get the answer. In a display hologram, we don't know where the pupil will be. In other words, the precise spot on the hologram that the observer looks through is not fixed. If the wavelength is changed, the image will shift, and the observer will be looking through a different part of the hologram in order to see the same image point. The same happens of course if the observer moves. That point is the center of our optical system.

So before using the elegant equations of Champagne, I had to use an iterative method (Newton's approximation) to find the principal ray. Only then could I apply the equations. This process has to be repeated for each point in a grid.

I will detail the mathematical background in another article soon.

Divisions by zero

I have caught a few cases where division by zero occurs, e.g. when `obj_dist = 0`. In this case I use the simple trick of giving it a very small distance. (I think I am showing that I am not a *real* programmer!) I appreciated any more cases of such errors, as well as suggestions of better ways to avoid them.

Lighting

This is set as a simple set of two lights, which can be changed.

Things to do

Short term

- Check angles for points in $-z$ region
- Group items better as macros in `holopov.inc`
- Improve data display
- Provide option for a solid looking cube, rather like ball and sticks, as now.
- Provide a user-friendly GUI. It is important that this is portable, and more or less device independent. I am presently trying out Revolution (`http://www.runrev.com`)

Longer term

- Show a 'real' object and its image, not just grid points or cubes
- Incorporate conventional optics (like a field lens) to modify image